EviCheck: Digital Evidence for Android

EviCheck: Digital Evidence for Android ⋆

Mohamed Nassim Seghir and David Aspinall

University of Edinburgh

Abstract. We present EviCheck, a tool for the verification, certification and generation of lightweight fine-grained security policies for Android. It applies static analysis to check the conformance between an application and a given policy. A distinguishing feature of EviCheck is its ability to generate *digital evidence*: a certificate for the analysis algorithm asserting the conformance between the application and the policy. This certificate can be independently checked by another component (tool) to validate or refute the result of the analysis. The checking process is generally very efficient compared to certificate generation as experiments on 20,000 real-world applications show.

1 Introduction and Related Work

Android security has been recently an active area of investigation and many tools for this purpose have emerged. Some of them rely on dynamic analysis like Aurasium [15], TaintDroid [8] and AppGuard [4]. Other ones are based on static analysis, like FlowDroid [2], ComDroid [6] and Apposcopy [9]. The last family of tools performs an exhaustive exploration of the application behaviour thanks to abstraction (over-approximation), which also leads to some imprecision. We are interested in this category (static analysis) as our aim is to certify the absence of bad behaviour. EviCheck complements these tools as it not only analyses applications but, returns a verifiable certificate attesting the validity of its result. The idea of associating proofs with code was initially proposed by Necula as *Proof-Carrying Code* (PCC) [11]. It has since been generalised to many forms of "certificate", not necessarily representations of proof in a logic. For example, abstract interpretations [1], validating intermediate steps in a compiler [10], etc. It was further used to provide bound guarantees on resource consumption [3, 5]. We call the generalised notion "digital evidence". The certificate returned by EviCheck broadens the PCC idea by encompassing lightweight forms of evidence specific to particular properties, e.g., program annotations tracking permissions or resource usage. Digital evidence can be independently checked to validate or refute the result of the analysis. A key point in PCC and related approaches is that the checking process is efficient compared to the generation one, in certain cases it could be 1000 times faster according to our experiments. Thus it may ultimately be carried out on the device itself at the point of installation, with the

⋆ This work was supported by EPSRC under grant number EP/K032666/1 "App Guarden Project".

generation process carried out once by the app supplier or app store. To release the user from the burden of writing anti-malware policies, EviCheck offers the option of inferring them automatically using constraint solving.

2 EviCheck's Main Ingredients

EviCheck has several components: policy language, verifier, checker and policy generator. The verification and certification processes are illustrated in Figure 1.

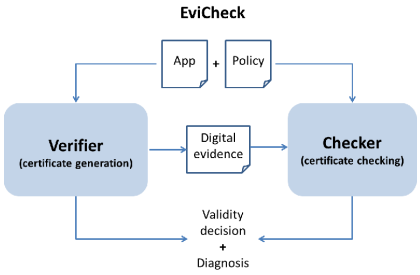

Fig. 1. The architecture of EviCheck and its main components

To illustrate the functionality of the various components of EviCheck, consider the code snippets and the associated graphical interface in Figure 2, which represent the audio recording app Recorder. The access to the recording device is carried out via object recorder (line 2).

```
1  public class Recorder extends Activity {
2    private MediaRecorder recorder = null;
3    ....
4    public void onCreate(....) {
5          ((Button) findViewById(Start))
6            .setOnClickListener(startClick);
7                ....
8          // startRecording();
9    }
10
11   private void startRecording() {
12         recorder = new MediaRecorder();
13         recorder.setAudioSource
14         (MediaRecorder.AudioSource.MIC);
15         recorder.setOutputFile(/* file name */);
16         ....
17         recorder.start();
18   }
19
20   private View.OnClickListener startClick
21           = new View.OnClickListener() {
22   public void onClick(View v) {
23         ....
24         startRecording();
25   }};
26   ....
27 }
```

Fig. 2. Code snippets and graphical interface of the Recorder app

At the creation phase (onCreate), a callback for a click event is associated with the button Start (line 5). Within the callback onClick, the method startRecording is invoked (line 24) which in turns calls recorder.setAudioSource and recorder.start to set the (on-device) microphone as a source and trigger the recording process. This app requires the permission RECORD_AUDIO which is associated with the API method setAudioSource.

2.1 Policy Language

The policy represents the specification that a user wants to check or a claim that a developer is stating about his app. In EviCheck, a policy consists of a set of rules obeying the grammar:

$$
\begin{aligned}
rule &:= H \ \ (: \mid \overset{or}{\vdots}) \ \ T \\
H &:= method \mid (CV|\neg CV)^+ \\
CV &:= \text{ENTRY\_POINT} \mid \text{ACTIVITY} \mid \text{SERVICE} \mid \text{RECEIVER} \\
&\quad \mid \text{ONCLICK\_HANDLER} \mid \text{ONTOUCH\_HANDLER} \mid LC \\
LC &:= \text{ONCREATE} \mid \text{ONSTART} \mid \text{ONRESUME} \mid \ldots \\
T &:= (id|\neg id)^*
\end{aligned}
$$

The head H of the rule represents a context in which the tail T should (not) be used. In the grammar, CV represents a context variable specifying the scope for methods to which the rule applies. For example, ENTRY_POINT: all entry points of the app, ACTIVITY: activity methods, ONCLICK_HANDLER: click event handlers, in addition to activity life-cycle callbacks such as ONCREATE, etc. The tail of the rule is a list of (negated) identifiers *id*'s or tags. We use permissions as tags, however tags can be supplied by the user as well. Referring to our running example, we can prohibit recording without user consent via the rule:

$$\text{ENTRY\_POINT}, \neg\text{CLICK\_HANDLER} \ : \ \neg\text{RECORD\_AUDIO} \tag{1}$$

It simply says: "*in all entry points, apart from click event handlers, the permission* RECORD_AUDIO *must not be used*". It is also possible to rule out another scenario of malicious behaviour, where a service reads the recorded file and sends it to a remote server, via the following rule:

$$\begin{aligned}\text{SERVICE} \ \overset{or}{\vdots} \ &\neg\text{INTERNET} \\ &\neg\text{READ\_EXTERNAL\_STORAGE}\end{aligned}$$

The superscript *or* indicates that this is an or-rule. Its semantics is that either permission INTERNET or READ_EXTERNAL_STORAGE can be used in a service component but not both. By default rules have an and-semantics. Due to space limitations, we do not provide a formal definition of the language semantics.

2.2 The Verifier

As Figure 1 shows, the verifier (certificate generator) takes an app and a policy as input and answers whether the policy is satisfied by the app and eventually outputs a certificate (digital evidence). If the policy is violated a diagnosis pointing

to the violated rules is returned. The verification algorithm consists of a reachability analysis which computes the transitive closure of the call graph with respect to permission usage. Referring to our running example, we start with the map below on the left side as we initially only know that the API method recorder.setAudioSource requires the RECORD_AUDIO permission as implemented by the Android framework. The analysis, which consists of a backward propagation, returns the map on the right side. In this case rule (1) is valid as the only entry points (underlined) we have are onClick and onCreate. If we uncomment line 8 in Figure 2, RECORD_AUDIO will be reachable from onCreate, hence violating the rule. The final map represents the certificate for the analysis.

setAudioSource : RECORD_AUDIO	setAudioSource : RECORD_AUDIO onClick : RECORD_AUDIO onCreate : startRecording : RECORD_AUDIO

2.3 The Checker

The checker takes as input a certificate (computed map), a policy and an app, and checks whether the certificate is valid (see Figure 1). It also checks whether the certificate entails the policy. If the certificate is invalid, a message referring to its first inconsistent entry is returned. Certificate checking is lighter than certificate generation as we do not need to compute a fix point (backward propagation). It suffices to go through each method and locally check if the associated set of tags includes all the tags associated with the functions it calls. This procedure has a linear complexity in the number of map entries (functions). It also has a constant space complexity as we are just performing checks without generating any information which needs to be stored.

2.4 Policy Generator

EviCheck is able to automatically infer anti-malware policies using constraint solving. To this end, we need a training set of malware and benign applications where each application is described by a set of properties (p's). For example p_i could be SERVICE : SEND_SMS, meaning that the permission SEND_SMS is used within a service component. A policy P excludes an application A if P contains a rule $\neg p_i$ and p_i belongs to the description of A. We want to find the properties $p_1, \ldots, p_k$ such that the policy composed of $\neg p_1, \ldots, \neg p_k$ allows a maximum of benign applications and excludes a maximum of malware. We use a a pseudo-Boolean solver, such as **Sat4j** [1], to solve this kind of optimisation problems where variable values are either 0 or 1 (true or false).

2.5 Technical Discussion

The call graph is the key representation on which our analysis relies. It is therefore essential that the generated call graph is as complete as possible. Java and

[1] http://www.sat4j.org/

object oriented languages in general have many features, such as method overriding, which makes the construction of an exact call graph (statically) at compile time undecidable. Therefore we over-approximate it using the *class hierarchy* approach [13] which permits to conservatively estimate the run-time types of the receiver objects.

Other Issues. Reflection is also a known issue for static analysis. A simple and conservative solution for this problem is to associate a tag t_{ref} with methods of the class java/lang/reflect/Method. We then use the tag t_{ref} to make the policy reflection-aware, e.g., $c : \neg t_{ref}$ to express that reflection should not be used in the context c. A similar solution is applicable for dynamic code loading by associating a tag t_{dyn} with methods of the class dalvik/system/DexClassLoader. Another framework-related challenge consists of modelling event handler callbacks. While the order of callback invocations can be over-approximated via a non-deterministic model, a more precise solution has been proposed in the literature [16]. However, adopting such an approach for EviCheck could incur an additional cost exceeding the one of the core analysis itself.

3 Implementation and Experiments

EviCheck is written in Python ($\sim$ 6000 lines) [12]. It uses Androguard [7] as back-end for parsing Android apps and has also an interface to Soot [14].

As mentioned previously, EviCheck has an option for the automatic generation of policies. In our experiments, we have first automatically generated a policy composed of 22 rules. An example of a rule is RECEIVER : ¬CAMERA, expressing that the camera should not be used in a broadcast receiver component. We then verified the generated policy against a representative set of more than **20,000** apps, from the Google Play store and Android observatory[2], ranging over different domains: games, social, etc. A snapshot of the results obtained with a typical desktop computer is illustrated in Figure 3.

App	G.T	C.T	#M	Policy
amazon	14.15	0.07	30498	
facebook	32.09	0.08	37106	
whatsapp	25.3	0.08	34143	✗
skype	11.9	0.06	25247	
twitter	17.4	0.07	32807	
...	...	...	...	...

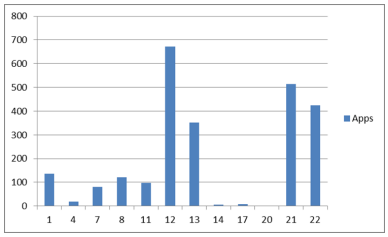

Fig. 3. In the table, G.T is the certificate generation time in seconds, C.T is the checking time and #M is the number of methods in an app. The symbol ✗ means that the policy is violated. The diagram on the right gives for each rule from the policy, identified by its number, the number of apps violating it from a set of 20,000 apps.

[2] https://androidobservatory.org/

In the table, symbol ✗ indicates that the policy is violated. Policy violation does not necessarily mean malicious behaviour; it can be used as an alarm trigger in a triage phase to filter out safe apps and to advise more careful scrutiny of the remaining ones. EviCheck can provide a detailed view pointing to the violated rule itself. The diagram on the right shows for each rule the number of apps violating it. The selected rules are numbered according to their appearance in the generated policy.

The checking time is less than 1 second for most of the apps and the ratio between the certificate generation and checking time is on average 70 but can in some cases reach 1000. This is encouraging as our aim is to carry out the checking process on the phone device, which is quite limited in terms of performance.

References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004.
2. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, page 29, 2014.
3. D. Aspinall and K. MacKenzie. Mobile resource guarantees and policies. In *CASSIS*, pages 16–36, 2005.
4. M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - enforcing user requirements on Android apps. In *TACAS*, pages 543–548, 2013.
5. G. Barthe, P. Crégut, B. Grégoire, T. P. Jensen, and D. Pichardie. The mobius proof carrying code infrastructure. In *FMCO*, pages 1–24, 2007.
6. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.
7. A. Desnos. Androguard. http://code.google.com/p/androguard/.
8. W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.
9. Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *FSE*, 2014. (to appear).
10. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
11. G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.
12. M. N. Seghir. Evicheck. http://groups.inf.ed.ac.uk/security/appguarden/tools/EviCheck/.
13. V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *OOPSLA*, pages 264–280, 2000.
14. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *CC*, pages 18–34, 2000.
15. R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *Presented as part of the 21st USENIX Security Symposium*, pages 539–552, Berkeley, CA, 2012. USENIX.
16. S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE*, 2015.